CONSTANTS

"Only thing constant is change"

– *Heraclitus*

constants belong to compile-time

unnamed constants

all basic literals are unnamed constant values

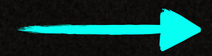
```
1
3.14
"hello"
true false
```

named constants

named constants will be replaced to their value in runtime

compile-time

```
const One int = 1
```



runtime

```
1
```

untyped constants

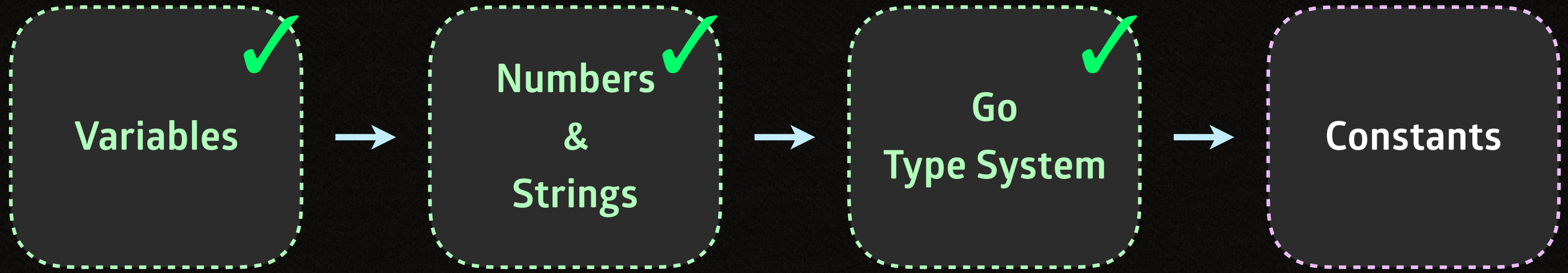
constants may or may not have a type

```
1
3.14
"hello"
true false
```

PART II

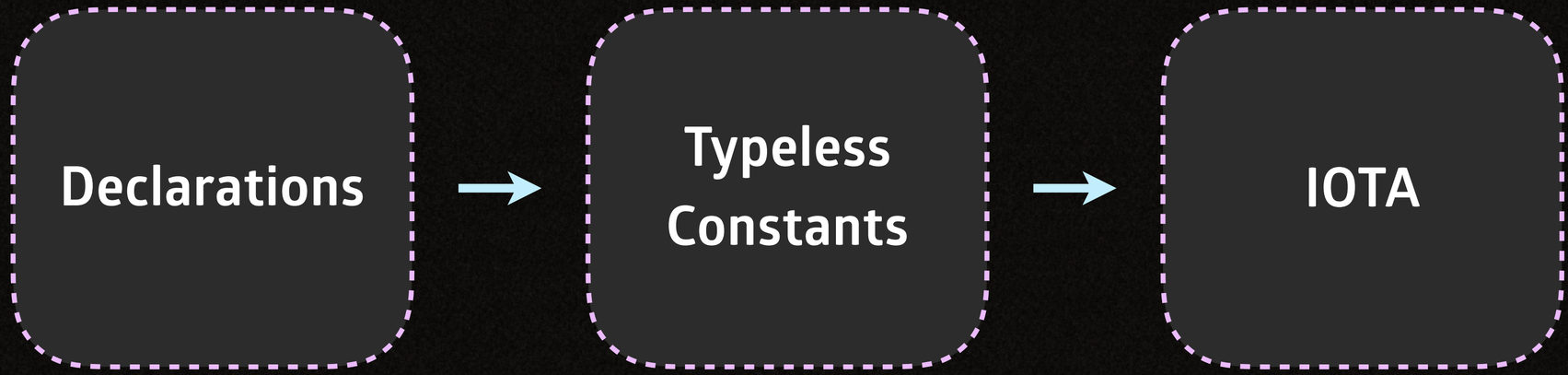
Fundamentals

Representing Data



CONSTANTS

You're going to learn about constants and untypedness



DECLARATION SYNTAX

Named Constants need to be declared

MAGIC VALUES

This program uses "magic numbers"

```go
func main() {
    cm := 100
    m  := cm / 100
    fmt.Printf("%dcm is %dm\n", cm, m)

    cm = 200
    m  = cm / 100
    fmt.Printf("%dcm is %dm\n", cm, m)
}
```

```
$ go run main.go
100cm is 1m
200cm is 2m
```

MAGIC VALUES

Let's get rid of the magic numbers

```go
func main() {
    const meters

    cm := 100
    m  := cm / meters
    fmt.Printf("%dcm is %dm\n", cm, m)

    cm = 200
    m  = cm / meters
    fmt.Printf("%dcm is %dm\n", cm, m)
}
```

```
$ go run main.go
ERROR: missing value in const declaration
```

MAGIC VALUES

Constants always have to be initialized to a value

```go
func main() {
    const meters int = 100

    cm := 100
    m  := cm / meters
    fmt.Printf("%dcm is %dm\n", cm, m)

    cm = 200
    m  = cm / meters
    fmt.Printf("%dcm is %dm\n", cm, m)
}
```

```
$ go run main.go
100cm is 1m
200cm is 2m
```

ERROR DETECTION

*Go **can't detect** runtime errors at the compile-time*

```go
func main() {
    total, numberOf := 5, 0

    fmt.Printf(total / numberOf)
}
```

```
$ go run main.go
RUNTIME ERROR: Integer divide by zero
```

ERROR DETECTION

*Go **can detect** the error at the compile-time*

```
func main() {
  const total     int = 5
  const numberOf  int = 0

  fmt.Printf(total / numberOf)
}
```

COMPILE-TIME ERROR: Division by zero

RULES

"Learn the rules like a pro, so you can break them like an artist."

– Pablo Picasso

RULES

You cannot change constant values (immutable)

```
func main() {
    const max int = 100
    max = 100
}
```

```
$ go run main.go
COMPILER ERROR: cannot assign to max
```

RULES

You cannot initialize a constant to a runtime value

```
func main() {
   const max int = math.Pow10(2)
}
```

function calls belong to runtime

```
math.Pow10(2) ⟶ 100
math.Pow10(3) ⟶ 1000
```

```
$ go run main.go
COMPILER ERROR:
... math.Pow10 is not a constant
```

RULES

You cannot initialize a constant to a runtime value

```
func main() {
    n := 2

    const max int = n
}
```

variables
belong to runtime

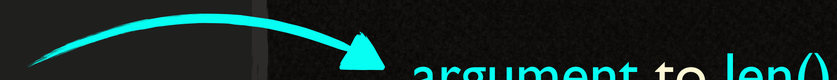
```
$ go run main.go
COMPILER ERROR:
... n is not a constant
```

RULES

You can initialize a constant using built-in function len

```
func main() {
    const max int = len("Hello")

    fmt.Println(max)
}
```




```
$ go run main.go
5
```

CONSTANT TYPES

You can declare constants using non-numeric types as well

CONSTANT TYPES

Constants don't have to be only numeric values

```
func main() {
   const min     int     = 1
   const pi      float64 = 3.14
   const version string  = "2.0.1"
   const debug   bool    = true

   fmt.Println(min, pi, version, debug)
}
```

```
$ go run main.go
1 3.14 2.0.1 true
```

CONSTANT TYPES

You don't have to declare the types of the constants

```go
func main() {
    const min     = 1
    const pi      = 3.14
    const version = "2.0.1"
    const debug   = true

    fmt.Println(min, pi, version, debug)
}
```

```
$ go run main.go
1 3.14 2.0.1 true
```

CONSTANT EXPRESSIONS

You can use expressions while initializing constants

```go
func main() {
    const min     = 1 + 1
    const pi      = 3.14 * min
    const version = "2.0.1" + "-beta"
    const debug   = !true

    fmt.Println(min, pi, version, debug)
}
```

```
$ go run main.go
2 6.28 2.0.1-beta false
```

MULTIPLE DECLARATION

You can declare multiple constants at once

MULTIPLE DECLARATION

You can declare multiple constants in the same line of code

```go
func main() {
    const min, max int = 1, 1000

    fmt.Println(min, max)
}
```

```
$ go run main.go
1 1000
```

MULTIPLE DECLARATION

You can declare multiple constants in a group

```go
func main() {
    const (
        min  int  = 1
        max  int  = 1000
    )

    fmt.Println(min, max)
}
```

```
$ go run main.go
1 1000
```

REPEATING CONSTANTS

Constants get their types and expressions from the previous constant

```
func main() {
   const (
      min  int = 1
      max  int = 1
   )
   fmt.Println(min, max)
}
```


It repeats
the previous constant's
type
and/or
expression

```
$ go run main.go
1 1
```

REPEATING CONSTANTS

Constants get their types and expressions from the previous constant

```
func main() {
    const (
        min  int = 1 + 1
        max  int = 1 + 1
    )
    fmt.Println(min, max)
}
```

It repeats
the previous constant's
type
and/or
expression

```
$ go run main.go
2 2
```

summary

declaring constants

```
const meters int = 100

const version string = "1.0.1"
```

exporting constants

```
const meters int = 100 // unexported

const Meters int = 100 // exported
```

constants can have different types

```
const (
    meters int     = 100
    feets  float64 = 1.5
)
```

you can't change a constant

```
const meters int = 100
meters = 200
```

constants prevent magic values

```
meters := feets * 0.3048
```

```
const feetInMeters = 0.3048
meters := feets * feetInMeters
```

constants need to be initialized

```
const meters int
```

```
const meters int = 100
```

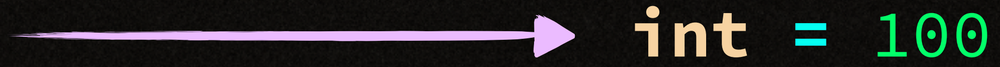
constants are only for compile-time

```
name := "Jake"
const person = name
```

```
const feet = math.Pow10(2)
```

grouped constants repeat the previous one

```
const (
    meters int = 100
    feets
)
```



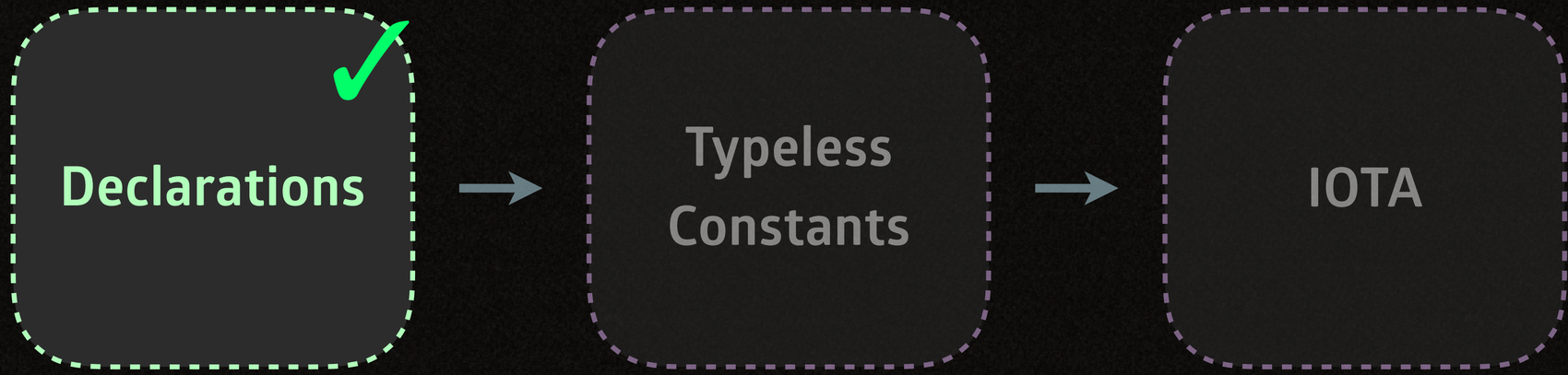
```
int = 100
```

constant expressions

```
const (
    meters int     = 100
    feets  float64 = meters / 3.2808
)
```

CONSTANTS

Congrats!



TYPELESS CONSTANTS

They don't have a type until one is needed

TYPELESS CONSTANTS

Previously....

```
func main() {
   const  min       = 1 + 1
   const  pi        = 3.14 * min

   fmt.Println(min, pi)
}
```

TYPELESS CONSTANTS

*Normally, you can't use a **float64** value and an **int** value together*

`float * int` ?

```
func main() {
    const min = 1 + 1
    const pi  = 3.14 * min

    fmt.Println(min, pi)
}
```

TYPELESS CONSTANTS

Untyped constants are like chameleons

min has no type

```
const min = 1 + 1
const pi  = 3.14 * min
```

pi has no type

When you declare a constant without a type
It becomes a untyped constant (typeless constant)

TYPELESS CONSTANTS

All basic literals are also typeless
They all are typeless constant values

```
const min = 1 + 1
const pi  = 3.14 * min
```

float literal (typeless)

int const (typeless)

TYPED vs TYPELESS

A constant with a type can only be used with a value of the same type

```
func main() {
    const min int = 42

    var i int
✓   i = min

    fmt.Println(i)
}
```

```
$ go run main.go
42
```

TYPED vs TYPELESS

A constant with a type can only be used with a value of the same type

```go
func main() {
    const min int = 42

    var f float64
    f = min

    fmt.Println(f)
}
```

```
$ go run main.go
```

COMPILER ERROR:

cannot use min (type **int**) as type **float64**

TYPED vs TYPELESS

An untyped numeric constant can be used with all numeric values together

```
func main() {
   const min = 42

   var f float64
✓  f = min

   fmt.Println(f)
}
```

typeless

min changes into a float64

```
$ go run main.go
42
```

TYPED vs TYPELESS

An untyped numeric constant can be used with all numeric values together

```
func main() {
   const min = 42

   var i int     = min
   var f float64 = min
   var b byte    = min
   var j int32   = min
   var r rune    = min

   fmt.Println(i, f, b, j, r)
}
```

min's type = int

min's type = float64

min's type = byte

min's type = int32

min's type = rune

```
$ go run main.go
42 42 42 42 42
```

TYPED vs TYPELESS

Let's analyze what happens behind the scenes

```
const min = 1
```

```
var b byte = min
```

TYPED vs TYPELESS

Go implicitly converts a typeless constant to a typed value

```
const min = 1

var b = byte(min)
```

DEFAULT TYPE

Every untyped constant has a default type

DEFAULT TYPES

Go converts a typeless constant to a typed value when a type is needed

```
const min int32 = 1

max := 5 + min
```

```
max := int32(5) + min
```

Go converts 5 to min's type automatically

max variable's type becomes **int32**

TYPED vs TYPELESS

Go implicitly converts the typeless constant to a typed value

```
const min = 1

max := 5 + min
```

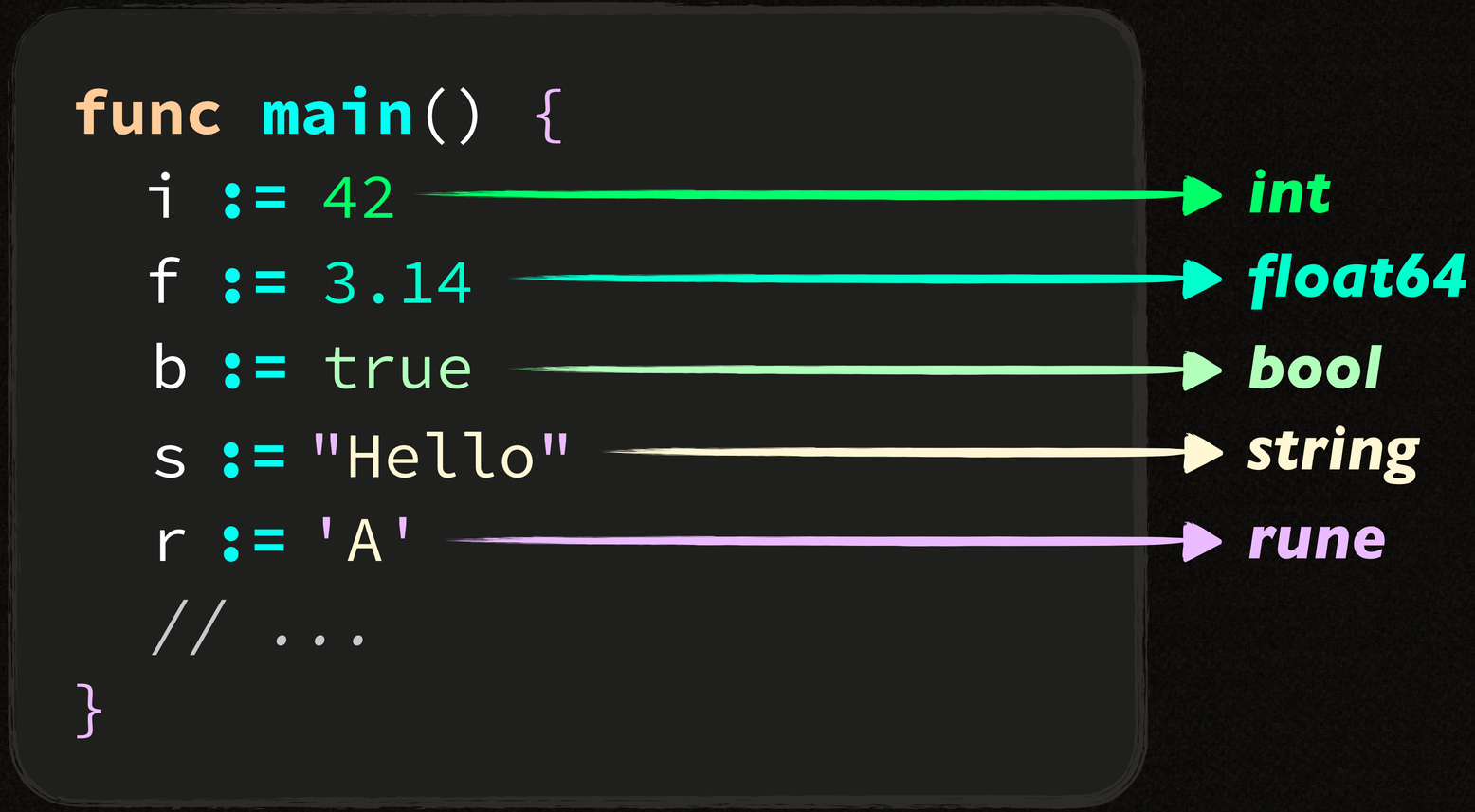
```
max := int(5) + int(min)
```

Go converts 5 and min to their default type

max variable's type becomes **int**

DEFAULT TYPE

An untyped constant has a default type



DEFAULT TYPE

What's the type of this expression and the variable?

```
func main() {
    i := 42 + 3.14
    // ...
}
```

DEFAULT TYPE

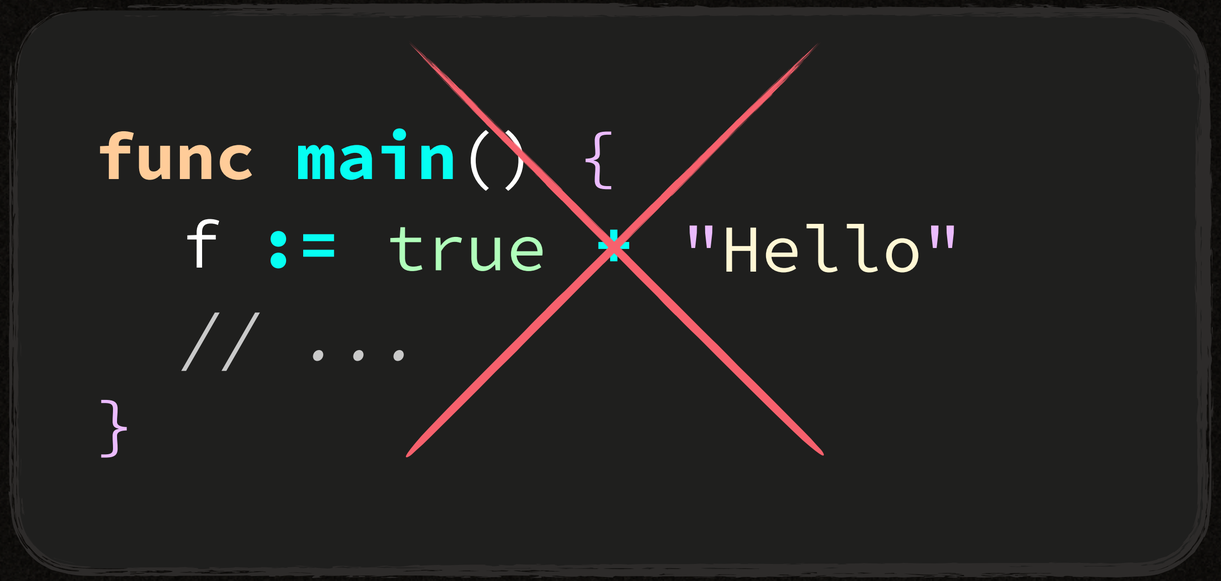
Go evaluates the expression then it converts the resulting typeless value to its default value

```
func main() {
  i := float64(42 + 3.14)
  // ...
}
```

the result of this expression is a typeless float value

DEFAULT TYPE

Typeless doesn't mean the utmost freedom



DEMONSTRATION

Let's take a look at a real-life constant usage

TYPE DEFINITION

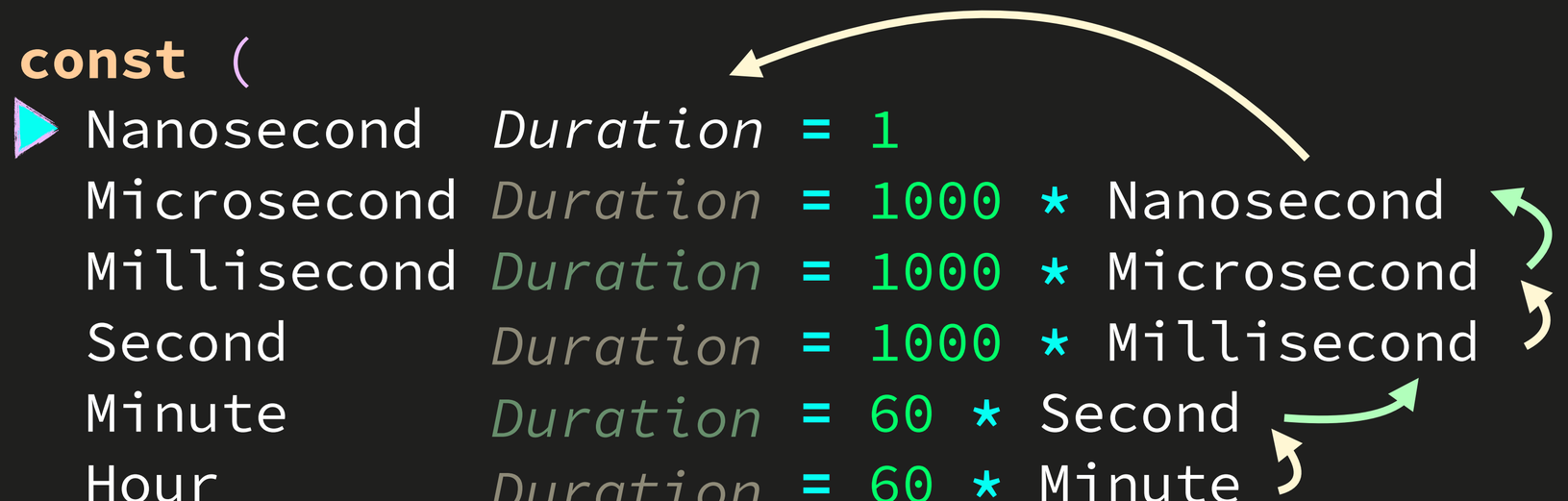
Duration is an exported and defined type of time package

```
type Duration int64
```

TIME.DURATION

Time package declares a few constants for measuring the duration between times

```
const (
    Nanosecond  Duration = 1
    Microsecond Duration = 1000 * Nanosecond
    Millisecond Duration = 1000 * Microsecond
    Second      Duration = 1000 * Millisecond
    Minute      Duration = 60 * Second
    Hour        Duration = 60 * Minute
)
```



- Why Nanosecond is declared with a type: It has methods + Type safety
- Microsecond and others get their type indirectly from Nanosecond

DEMONSTRATION

typeless integer literal

```
func main() {
    t := time.Second * 10
    fmt.Println(t)

    i := 10
    t = time.Second * i
}
```

time.Duration int

```
$ go run main.go
10s
```

DEMONSTRATION

```
func main() {

   fmt.Print(t)

   i := 10
   t  = time.Second * i
}
```

DEMONSTRATION

```go
func main() {
    i := 10
    t  = time.Second * time.Duration(i)

    fmt.Println(t)
}
```

✓

i becomes time.Duration which is int64

```
$ go run main.go
10s
```

DEMONSTRATION

```go
func main() {
  const c = 10
  t  =  time.Second  *  c

  fmt.Println(t)
}
```

✓

c
implicitly
becomes
time.Duration
which is int64

```
$ go run main.go
10s
```

FEET TO METERS

REFACTORING

FEET TO METERS

EXERCISE

FEET TO METERS

SOLUTION

summary

a constant without a type is typeless

```
const meters = 100
```

basic literals are untyped constants

```
3.14
100
"Hello"
true false
'X'
```

typeless constants are flexible

```
const meters = 100

feets := meters / 3.2808
```

default types

```
100   → int
3.14  → float64
"Hi"  → string
 "X"  → rune
true  → bool
false → bool
```

CONSTANTS

Congrats!

IOTA

Built-in constant number generator

IOTA

iota is a built-on constant generator which generates ever increasing numbers

```go
func main() {
    const (
        monday      = 0
        tuesday     = 1
        wednesday   = 2
        thursday    = 3
        friday      = 4
        saturday    = 5
        sunday      = 6
    )

    fmt.Println(monday, tuesday, wednesday, thursday, friday,
        saturday, sunday)
}
```

```
$ go run main.go
0 1 2 3 4 5 6
```

IOTA

iota is a built-on constant generator which generates ever increasing numbers

```
func main() {
   const (
      monday      = iota
      tuesday     = iota
      wednesday   = iota
      thursday    = iota
      friday      = iota
      saturday    = iota
      sunday      = iota
   )

   fmt.Println(monday, tuesday, wednesday, thursday, friday,
      saturday, sunday)
}
```

0
1
2
3
4
5
6

```
$ go run main.go
0 1 2 3 4 5 6
```

IOTA

You can use expressions with iota. So, the other constants will repeat the expressions

```go
func main() {
    const (
        monday      = iota + 1
        tuesday     = iota + 1
        wednesday   = iota + 1
        thursday    = iota + 1
        friday      = iota + 1
        saturday    = iota + 1
        sunday      = iota + 1
    )

    fmt.Println(monday, tuesday, wednesday, thursday, friday,
        saturday, sunday)
}
```

1
2
3
4
5
6
7

```
$ go run main.go
1 2 3 4 5 6 7
```

BLANK-IDENTIFIER

Black-Hole is on the scene again!

BLANK-IDENTIFIER

Let's declare a few constants for representing timezones

Timezones

```
EST  -5
MST  -7
PST  -8
```

```
$ go run main.go
-5 -7 -8
```

```go
func main() {
    const (
        EST = -5
        MST = -7
        PST = -8
    )

    fmt.Println(EST, MST, PST)
}
```

BLANK-IDENTIFIER

Which formula do I need to initialize constants with correct timezone values?

Timezones

EST	-5	✓
MST	-7	✗
PST	-8	✗

```
$ go run main.go
-5 -6 -7
```
✗

```
func main() {
    const (
        EST = -(5 + iota)
        MST
        PST
    )

    fmt.Println(EST, MST, PST)
}
```

✓	-5	-(5+0)=-5
✗	-6	-(5+1)=-6
✗	-7	-(5+2)=-7

BLANK-IDENTIFIER

Which formula do I need to initialize constants with correct timezone values?

Timezones

EST **-5** ✓
MST **-7** ✓
PST **-8** ✓

```
$ go run main.go
-5 -7 -8
```
✓

```
func main() {
   const (
      EST = -(5 + iota)
      _
      MST
      PST
   )

   fmt.Println(EST, MST, PST)
}
```

✓ **-5** -(5+0)=-5
-6 -(5+1)=-6
✓ **-7** -(5+2)=-7
✓ **-8** -(5+3)=-8

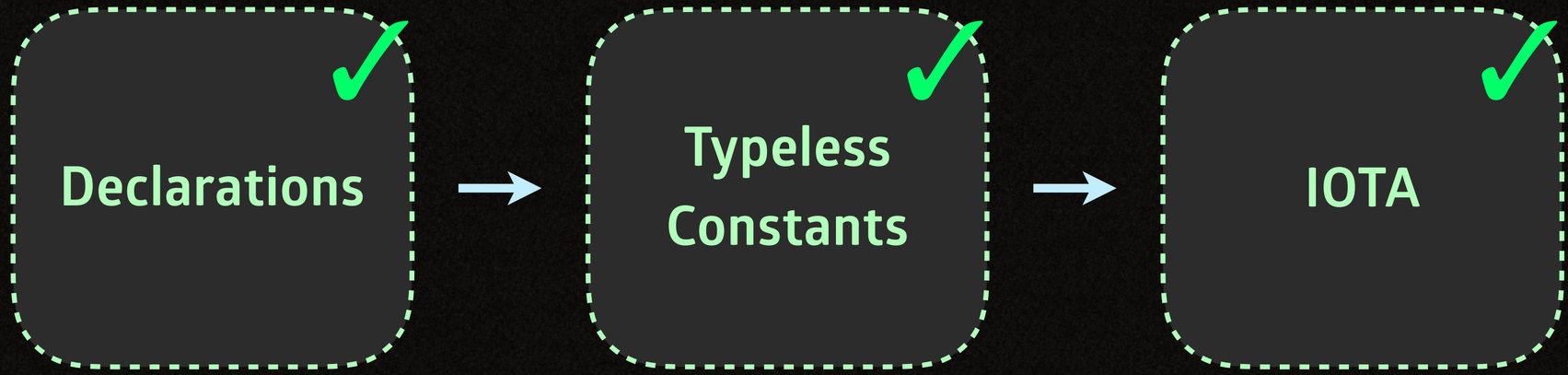
summary

iota is a constant number generator

blank-identifier

CONSTANTS

Congrats!



NAMING THINGS

there only two hard things in computer science

cache invalidation and

naming things

- Phil Karlton

NOT IDIOMATIC

Unnecessarily verbose

```
func Read(buffer *Buffer, inBuffer []byte) (size int, err error) {
    if buffer.empty() {
        buffer.Reset()
    }

    size = copy(
      inBuffer,
      buffer.buffer[buffer.offset:])

    buffer.offset += size
    return size, nil
}
```

IDIOMATIC

Concise and idiomatic

```
func Read(b *Buffer, p []byte) (n int, err error) {
    if b.empty() {
        b.Reset()
    }

    n = copy(p, b.buf[b.off:])

    b.off += n

    return n, nil
}
```

ABBREVIATIONS

Common abbreviations used in Go

```
var s string                // string
var i int                   // index
var num int                 // number
var msg string              // message
var v string                // value
var val string              // value
var fv string               // flag value
var err error               // error value
var args []string           // arguments
var seen bool               // has seen?
var parsed bool             // parsing ok?
```

ABBREVIATIONS

Common abbreviations used in Go

```
var buf []byte          // buffer
var off int             // offset
var op int              // operation
var opRead int          // read operation
var l int               // length
var n int               // number or number of
var m int               // another number
var c int               // capacity
var c int               // character
var a int               // array
var r rune              // rune
var sep string          // separator
```

ABBREVIATIONS

Common abbreviations used in Go

```
var src int                 // source
var dst int                 // destination
var b byte                  // byte
var b []byte                // buffer
var buf []byte              // buffer
var w io.Writer             // writer
var r io.Reader             // reader
var pos int                 // position

...list goes on and on...
```

Why naming is important?

★★★

Critical for Readability = Maintainability

★★★

Use the first few letters of the words


```
var fv string // flag value
```

Use fewer letters in smaller scopes

```
var bytesRead int // number of bytes read
```

```
var n int // number of bytes read
```

Use the complete words in larger scopes


```
package file
var fileClosed bool
```

Use mixedCaps like this


```
type PlayerScore struct
```

Use all capitals for acronyms

```
var localAPI string
```

```
var localApi string
```

Do not stutter


```
player.PlayerScore
```

❌

```
player.Score
```

✓

★★★

Do not use under_scores oR LIKE_THIS

★★★

```
const MAX_TIME int
```

❌

```
const MaxTime int
```

✓

```
const N int
```

✓